The DINO Parallel Programming Language

Matthew Rosing, Robert B. Schnabel,
and Robert P. Weaver

CU-CS-457-90 April 1990

Department of Computer Science
Campus Box 430
University of Colorado,
Boulder, Colorado, 80309 USA

This research was supported by AFOSR grant AFOSR-85-0251, and NSF Cooperative Agreement DCR-8420944.

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
APR 1990		**00-04-1990 to 00-04-1990**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
The DINO Parallel Programming Language	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of Colorado at Boulder,Department of Computer Science,Boulder,CO,80309-0430**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**48**	

Abstract

DINO (DIstributed Numerically Oriented language) is a language for writing parallel programs for distributed memory (MIMD) multiprocessors. It is oriented towards expressing data parallel algorithms, which predominate in parallel numerical computation. Its goal is to make programming such algorithms natural and easy, without hindering their run-time efficiency. DINO consists of standard C augmented by several high level parallel constructs that are intended to allow the parallel program to conform to the way an algorithm designer naturally thinks about parallel algorithms. The key constructs are the ability to declare a virtual parallel computer that is best suited to the parallel computation, the ability to map distributed data structures onto this virtual machine, and the ability to define procedures that will run on each processor of the virtual machine concurrently. Most of the remaining details of distributed parallel computation, including process management and interprocessor communication, result implicitly from these high level constructs and are handled automatically by the compiler. This paper describes the syntax and semantics of the DINO language, gives examples of DINO programs, presents a critique of the DINO language features, and discusses the performance of code generated by the DINO compiler.

1. Introduction

DINO (DIstributed Numerically Oriented language) is a language for writing parallel numerical programs for distributed memory multiprocessors. By a distributed memory multiprocessor we mean a computer with multiple, independent processors, each with its own memory, but with no shared memory or shared address space, so that communication between processors is accomplished by passing messages. Examples include MIMD hypercubes, and networks of computers used as multiprocessors.

It is generally harder to design an algorithm for a multiprocessor than for a serial machine, because the algorithm designer has more tasks to accomplish. First, the algorithm designer must divide the desired computation among the available processors. Second, the algorithm designer must decide how these processes will synchronize. In addition, on a distributed memory multiprocessor, the algorithm designer must consider how data should be distributed among the processors, and how the processors should communicate any shared information.

The goal of DINO is to make programming distributed memory parallel numerical algorithms as easy as possible, without hindering efficiency. We believe that the key to this goal is raising the task of specifying algorithm and data decomposition, interprocess communication, and process management to a higher level of abstraction than is provided in many current message passing systems. DINO accomplishes this by providing high level parallel constructs that conform to the way the algorithm designer naturally thinks about the parallel algorithm. This, in turn, transfers many of the low-level details associated with distributed parallel computation to the compiler. In particular, details regarding message passing, process management, and synchronization are no longer necessary in the code that the programmer writes, and associated efficiency considerations are addressed by the compiler.

This high level approach to distributed numerical computation is feasible because so many numerical algorithms are highly structured. The major data structures in these algorithms are usually arrays, and sometimes trees. In addition, the algorithms usually exhibit data parallelism, where at each stage of the computation, parallelism is achieved by dividing the data structures into pieces, and performing similar or

identical computations on each piece concurrently. DINO is mainly intended to support such data parallel computation. DINO also provides some support for functional parallelism.

The basic approach taken in DINO is to provide a top down description of the distributed parallel algorithm. The programmer first defines a virtual parallel machine that best fits the major data structures and communication patterns of the algorithm. Next, the programmer specifies the way that these major data structures will be distributed, and possibly replicated, among the virtual processors. Finally, the programmer provides procedures that will run on each virtual processor concurrently. Thus the basic model of parallelism is Single Program Multiple Data, although far more complexity is possible. Most of the remaining details of parallel computation and communication are handled implicitly by the compiler. A key component in making this approach viable has been the development of a rich mechanism for specifying the mappings of data structures to virtual machines, along with the efficient implementation of these mappings and the resultant communication patterns in the compiler.

Sequential code in a DINO program is written in standard C; DINO is a superset of C. Only the parallel constructs in DINO, and a few related additions, are new. We have chosen to build DINO on top of C because C is widely used and is available on our target parallel machines, because its structured form fits well with our parallel approach, because its dynamic data structures are useful to many numerical applications, and because we have many tools available for implementing a C compiler. However the DINO language as described in this paper could have been built on top of almost any imperative sequential language, including FORTRAN.

Implicit in our approach to parallel programming is the belief that given the current state of knowledge in this area, a language cannot completely hide the distributed memory parallel machine from the programmer if the result is to be programs that run efficiently. In conjunction, we believe that most parallel algorithm designers do have an idea of how to break up the data structures and how to act upon them concurrently in order to achieve efficient parallel performance. In DINO, we are trying to provide just enough high level parallel structure to allow the programmer to specify this information, while making the compiler do as much of the tedious work as possible.

To our knowledge, there is relatively little other research that has led to implemented languages for distributed parallel numerical computation. C* [RSt87] and Kali [MvR89, KMvR90] are probably the closest languages we have found to DINO, although DINO was developed entirely independently of both of them. C* is a C superset used to program the (SIMD) Connection Machine. Its domain and selection statement features are similar to DINO's environment structures and composite procedures, respectively; however, its SIMD orientation gives it a considerably different philosophy and flavor from DINO. One prominent difference is C*'s local view of data structures as opposed to DINO's global view. Kali is a high level language for data parallel, SPMD computation. It contains constructs for virtual parallel machines and distributed data structures that are similar to DINO's (but see next paragraph). Its programming model, however, is considerably more restrictive than DINO's, allowing communication only upon entering and leaving "forall" loops. Another parallel language that has been implemented on distributed memory multiprocessors, Linda [GCC85], is designed around a shared memory paradigm, the "tuple space". Due to this approach, its constructs and philosophy are very different from DINO's. For example, the mapping of data to processors is not specified by the Linda programmer and must be optimized at run time. Pisces [Pr87] and the Cosmic Environment [SSS88] are examples of systems that also support distributed parallel numerical applications, but using a lower level of support for parallelism than DINO. Several languages, such as Spot [So90] and Apply [HWW89], support specific application areas where a C*-like local view of the computation is appropriate; our research aims to support a broader class of computations and this paradigm is not sufficient. Our work has been influenced considerably by the Force [Jo87], which supports data parallel computation on shared memory multiprocessors.

In recent years, the concept of distributed data, which is key to DINO, has been considered by several research projects, including [SBB87], [CK88] and [MvR89]. An important difference in the DINO version is that one to many mappings of data to processors, as well as one to one mappings, are supported. This turns out to be important for implicit interprocess communication. Suspense [RuW88], which is oriented to grid-based differential equations algorithms, shares some of these data partitioning and overlapping approaches of DINO. The concept of virtual machines is also found in Structured Process [LW85]. Finally, some earlier related research oriented primarily to SIMD computation includes

[KS85], [Re84].

The remainder of this paper is organized as follows. Section 2 gives a brief informal overview of the DINO language, while Section 3 gives a complete, more formal description. Section 4 presents three simple DINO examples, and uses them as a basis for a critique of the DINO language features. Section 5 briefly summarizes the current status of the language and areas for related future research. Preliminary discussions of the DINO project can be found in [RSc87, RSW88], and a moderate-sized example is presented in [RSW89].

2. DINO Overview

2.1. Environments

To create a parallel DINO program, the programmer first declares a structure of environments that best fits the number of processes and the communication pattern between processes in the parallel algorithm. This structure can be viewed as a virtual parallel machine constructed for the particular algorithm. In our experience, the most common structures of environments are one, two, and higher dimensional arrays of processes. This is because the parallelism in numerical algorithms is often derived from the partitioning of physical space into neighboring subregions, or from partitioning matrices or vectors.

Each environment in a structure of environments consists of data and procedures. It may contain multiple procedures, but only one process in an environment may be active at a time. Each environment in a single structure contains identical procedures and data structures, although the contents of the data structures, and the sequence of statements executed at run time, can differ.

Each DINO program also contains a "host" environment, a scalar environment that acts as the master process. The following declarations declare "host" to be the host environment and "node" to be a one dimensional array of 32 environments:

```
environment host {
    <body of environment declaration>
}
environment node [32] {
    <body of environment declaration>
}
```

2.2. Composite Procedures

A composite procedure is a set of identical procedures, one residing within each environment of a structure of environments. The parameters of a composite procedure typically are distributed variables (discussed below). A composite procedure call causes each instance of the procedure to execute concurrently, utilizing the portion of the distributed parameters and other distributed data structures that are mapped to its environment. This results in a ''single program, multiple data'' form of parallelism.

An example of a composite procedure declaration is:

```
composite MatVecMult(in M, in v, out a)
    <parameter declarations>
  {
   <body of procedure>
  }
```

The keywords ''in'' and ''out'' specify that the distributed parameters are value and result parameters, respectively. If there is no keyword, the default is value-result. This difference from the C parameter passing mechanism, which pertains only to parameters of composite procedures, is necessary because C's mechanism of passing a pointer to an object that is changed is not appropriate for remote procedure calls on a distributed memory multiprocessor.

An example of a composite procedure call corresponding to the above declaration is:

```
MatVecMult(Min[][], vin[], aout[])#;
```

The empty brackets (''[]'') mean that the entire corresponding dimension is included. The pound sign (''#'') is used in DINO when an action involves a remote environment.

2.3. Distributed Data

DINO encourages the programmer to take a global view of data structures that are distributed among multiple processors and operated upon concurrently. The programmer does this by declaring a data structure as distributed data, and then specifying the manner in which that data structure is mapped to the underlying virtual machine given by the programmer-defined structure of environments. A mapping of data to environments may be one to one, one to many, or one to all. These mappings determine how the environments access and share the data, and are the key to making interprocess communication natural and implicit. Distributed variables are referenced using their global names and subscripts, as described below.

The following distributed data declarations distribute the matrix M and the vectors v and a used as parameters in Section 2.2 to the structure of environments *node* defined in Section 2.1:

```
float distributed M[N][N] map BlockCol;
float distributed v[N] map all;
float distributed a[N] map Block;
```

The names "BlockCol," "all," and "Block" come from DINO's library of pre-defined mapping functions. Their meanings are the obvious ones: N/P columns of M are mapped to each of the P node environments, the entire vector v is mapped to each node environment, and N/P elements of a are mapped to each node environment. Thus, the mappings of M and a are partitions, with only one copy of any given element existing among all the node environments. In contrast, the vector v is replicated, with a local copy of v existing in each node environment.

Distributed data can be used to implicitly generate interprocess communication in two ways. (In both cases, blocks of data can be sent in a single operation.) One is by using a distributed variable as a parameter in a composite procedure call, as illustrated above. At procedure invocation, the values of distributed variables declared as input parameters are distributed to all environments to which they are mapped. Upon return from the composite procedure, the values of distributed variables declared as output parameters are collected from their environments (variables that are mapped to multiple environments

are collected from their ''home'' environments, see below).

Communication also results from ''remote'' access to distributed variables. In general, distributed data can be accessed either locally or remotely. A local access, which uses standard syntax, affects just the local copy and is the same as any standard reference to a variable. A remote access, which uses the variable name followed by a # sign, can be used with a variable element that is mapped to multiple environments. A remote assignment to a distributed variable causes a message containing its current value to be sent to all other environment(s) to which that variable has been mapped. A remote read of a distributed variable receives such a message (as described in more detail below) and updates the variable's value. Examples of a remote read and write are:

```
Temp = T[id]# + 3;
T[id]# = Temp - 3;
```

In the first case, a new value of *T*[*id*] is received and stored, and is used in calculating the new value of *Temp*. In the second case, the value of *Temp* - 3 is assigned to the local copy of *T*[*id*] and is also sent to all other environments to which *T*[*id*] is mapped.

In the default case DINO distributed variables are ''synchronous'', and communication using such variables is deterministic. A remote read causes the local copy of the variable to be overwritten with the first (least recent) value of that variable that has been received from its ''home'' environment but not yet utilized; if no new value is present, it blocks until one is received. (The concept of a home environment is explained in Section 3.4.1; basically, a replicated variable has a primary environment, from which new values of the variable are generated, and one or more secondary environments, at which new values are received. As with most constructs described in this overview section, other options exist and may be specified explicitly.) A distributed variable may also be declared to be asynchronous by using the key-words ''asynchronous distributed'' instead of ''distributed'' in its declaration. The execution of a remote write is unaffected. A remote read of an asynchronous variable causes it to overwrite its local copy with the latest value that has been received since the last remote read; if no value has been received, it retains

its current local value, and does not block. Thus its execution may be non-deterministic. These constructs allow experimentation with synchronous and asynchronous versions of the same algorithm by simply changing a variable declaration.

DINO provides a library containing many commonly used mapping functions, including the ones used above, and a general facility that allows the user to declare a very large set of data mappings (Section 3.4.1). These include the ability to specify block mappings, wrap mappings of arbitrary width, and arbitrary degrees of overlap, in each axis of the distributed data structure.

3. The DINO Language

In this section we present a more complete description of the syntax and semantics of the language. We use a combination of discussion, examples, and EBNF notation [WG84] to describe the additions to C that make up DINO. An EBNF specification for the DINO extensions to the C language, and an explanation of EBNF notation, can be found in Appendix A. (Note that this specification includes several EBNF constructs for standard C that are unchanged, namely CONSTANT-EXPRESSION, EXPRESSION LIST, FUNCTION-BODY, and IDENTIFIER; several EBNF constructs for standard C that are expanded, namely DECLARATOR, EXPRESSION, FUNCTION-DEFINITION, PRIMARY, AND STATEMENT; and one EBNF construct for standard C, DATA-DEFINITION, whose definition is unchanged but which contains a part, DECLARATOR, that is expanded. In all cases, only the new definitions are included here.) For a BNF specification for C, see [KR78].

3.1. Program Structure

```
PROGRAM ::= ( ENVIRONMENT | MAPPING_FUNCTION | DATA_DEFINITION )+ .
```

A program consists of one or more environment declarations, zero or more mapping function declarations, and zero or more (distributed) data declarations. One of the environment declarations must be defined as a scalar environment with the name ''host''. This environment must contain a procedure

named ''main'', where execution starts. The remaining environment declarations generally define structures of environments that contain composite procedures which are invoked from the host.

Distributed data structures that are declared at the program level are mapped to one or more environment structures in the program, as specified by their mapping functions. (Ordinary data declarations can also be made at this level. If they are, independent copies are instantiated on every environment in the program.) Mapping functions that are declared at the program level are accessible to all parts of the program.

3.2. Environments

```
ENVIRONMENT ::=
    'environment' IDENTIFIER DIMENSION* '{' EXTERNAL_DEFINITION+ '}'.

DIMENSION ::= '[' EXPRESSION [ ':' IDENTIFIER ] ']'.

EXTERNAL_DEFINITION ::=
    FUNCTION_DEFINITION | DATA_DEFINITION | MAPPING_FUNCTION.
```

A structure of environments provides a virtual parallel machine, and a mechanism for building parallel algorithms in a top down fashion. An environment may contain composite procedures, standard C functions, distributed data declarations, standard C data declarations, and mapping functions. Each environment within a given structure contains the same procedures, functions, and C data declarations, and shares the same distributed data declarations.

A structure of environments may be any single or multiple dimensional array. For example, the declaration

```
environment grid [N:xid] [M:yid] { ... }
```

specifies an N times M array of virtual processors named ''grid''. In order to give each environment within a structure an identity that can be used in calculations, the programmer can declare constants that will contain the subscripts identifying that particular environment. In the above example, these are ''xid'' and ''yid''. Most often these are used to refer to that environment's portion of a distributed data structure, for example a matrix element A [*xid*][*yid*].

A DINO program may contain any number of environment declarations. Typically only one, the host, is scalar, and one or more are arrays. Multiple structures of environments may be used when successive phases of a computation map naturally onto different parallel machines (in this case, generally only the environments in one structure become active at once), or in functionally parallel programs (in this case the environments in multiple structures become active at the same time, see Section 3.3).

An environment can contain an arbitrary number of composite procedures and standard C functions. However only one task, or thread of control, may be active in an environment at a time. In addition, the only way to start a task executing in an environment other than the host is to call a composite procedure which resides in that environment. These two facts imply that there can not be any composite procedures in the host environment because procedure ''main'' is always active.

A procedure executing in one environment can not directly access data in another environment. Procedures in two environments can exchange information only if there is cooperation between the procedures, namely a remote read and write of a distributed variable, or the use of a reduction operator on a distributed variable (Sections 3.4, 3.5). (We note that an option that relaxes these rules somewhat is likely to be available in the future — see Section 5 — but that there are significant tradeoffs between relaxing these rules in general and providing a language that is applicable and efficient for a wide variety of applications.) These semantics and the single task semantics described above imply that there is is no hidden shared-memory emulation in DINO, and also results in DINO programs being deterministic unless asynchronous distributed variables or explicit environment sets are used (see Section 3.4).

Structures of environments are treated as blocks with respect to scope. Ordinary data that is declared within an environment structure is copied to every environment in that structure. Copies of ordinary data on separate environments have no relation to each other and can only be accessed within their own environments. Distributed data is treated as described in Section 3.4. The one exception to this scoping rule is composite procedures, which are always in the topmost scope. Thus composite procedures may be called from any environment except their own, i.e. recursive calls are not allowed.

The DINO compiler creates one process for every environment in each structure, and maps these processes statically onto the actual parallel machine. This is done in a manner that attempts to optimize the distance, on the parallel computer, between contiguously indexed environments in each structure, using standard techniques [In87]. The mapping also attempts to optimize load balance, as follows. If any structure of environments contains as many or more environments than there are actual processors, then the environments are partitioned evenly over the entire parallel machine. (When a composite procedure is invoked in a structure with more environments than processors, multiprogramming occurs. This style of programming does not lead to optimal efficiency, but may in future versions of DINO — see Section 5.) If there are two or more structures of environments that together contain no more environments than the total number of processors, then each environment is assigned to a unique processor.

3.3. Composite Procedures

declaration:

```
FUNCTION_DEFINITION ::=
    'composite' IDENTIFIER '(' [COMP_PARAMETER_LIST] ')' FUNCTION_BODY .

COMP_PARAMETER_LIST ::= ( ['in' | 'out'] IDENTIFIER ) || ',' .
```

call:

```
STATEMENT ::=
    IDENTIFIER '(' [EXPRESSION_LIST] ')' '#' [ '{' ENV_EXP '}' ] [ '::' STATEMENT ] .

ENV_EXP ::= EXPRESSION .
```

Composite procedures are used to implement concurrency. A composite procedure consists of multiple copies of the same procedure, one residing within each environment of a structure of environments. Calling the composite procedure invokes all of these procedures at the same time. Typically, each procedure works on a different part of some distributed data structure(s), resulting in a single program, multiple data (SPMD) form of parallelism. These distributed data structures may either be defined in the structure of environments or globally, or may be parameters to the composite procedure. Each procedure may also contain standard local data. We will first describe the various parts of composite procedure

declaration and invocation statements, and then the order of events that occur when a composite procedure is invoked.

Examples illustrating a composite procedure and its invocation were given in Section 2.2. The formal parameters of a composite procedure may be distributed variables or standard variables whose scope includes the composite procedure. Formal parameters that are distributed variables may be preceded by the keywords ''in'' (call-by-value), ''out'' (call-by-result), or no keyword (call-by-value/result). Formal parameters that are standard variables must be preceded by ''in'', and are input parameters that are replicated in all the environments on which the composite procedure is called.

A composite procedure call has the same syntax as for ordinary C functions, except that arrays or subsections of arrays can be used (see Section 3.6), remote references to distributed parameters can be actual parameters, and the parameter list is followed by a # sign. All actual parameters corresponding to result and value/result parameters must refer to variables. Unlike C, DINO checks the number and types of actual parameters against the formal parameters in a composite procedure call. Composite procedures do not return a value.

When a composite procedure is called, the invocation can be limited to a subset of the environments in the environment structure on which it is defined. This is accomplished using the optional ENV_EXP following the # sign, which returns a subset of environments on which the procedure is to be invoked (the ''active-set''). The form of ENV_EXP is one or more environment names (or ranges — see Section 3.6) connected by set union or difference operators ''+'' or ''-''. If this expression is not given, then the composite procedure is invoked on all of the environments in the structure.

Functional parallelism is achieved in DINO by utilizing the optional '' :: STATEMENT'' construct following a composite procedure call (this definition is recursive, see the EBNF specification above). This STATEMENT is executed concurrently with the composite procedures specified by the call. It can be either another composite procedure call utilizing a different structure of environments (or a disjoint subset of the same structure), or a standard C statement that is executed on the host. The program blocks

until both the composite procedure and the concurrent statement complete execution.

The keyword ''caller'', when used within a composite procedure, is the environment which invoked that composite procedure. This may be used when a composite procedure is executing concurrently with code on the invoking environment, as described in the previous paragraph, and wants to communicate with the procedure that invoked it, via explicit reads and writes of global distributed data (see Section 3.4.2).

Now we describe in detail the events that occur when a composite procedure is invoked. First, the ENV_EXP expression is evaluated if it is given. This defines an active-set of environments as described above. Only procedures and environments in the active-set are used in the rest of the sequence. Second, all of the actual parameters that correspond to value and value/result formal parameters are evaluated. If any of these actual parameters contains a ''#'' operator then a remote read is executed to get the value (see Section 3.4.2). Third, the values of the actual parameters are assigned to the formal parameters and sent to the appropriate environments. For formal parameters that are not defined as ''distributed'', the actual parameter value is sent to each environment in the active-set. If the formal parameter is defined as ''distributed'' then the actual parameter value is distributed to each environment that it is mapped to (but limited to the active-set). Fourth, the procedure is actually called at each environment in the active set. At the same time, the concurrent STATEMENT, if any, is executed in the calling environment (possibly resulting in one or more additional composite procedure calls on different environments). Fifth, after all these procedures/STATEMENT have completed executing, the result and value/result parameters are returned. Each element of a result parameter is returned from exactly one environment, its ''home'' environment (see Section 3.4.1). If any of these actual parameters contains a ''#'' operator then a remote write is be executed. Finally, the calling environment continues execution after the concurrent STATEMENT.

3.4. Distributed Data

Distributed data is used to map global data structures onto the structures of environments in a DINO program. It provides the mechanism for communication between environments, which is accomplished by using distributed variables as parameters to composite procedures, and by remote accesses to distributed variables. It also allows the programmer to maintain a global view of these data structures, and joins with composite procedures to produce an SPMD model of computation.

Currently, the types of data structures that can be distributed are arrays of any dimension. DINO provides a rich mechanism for specifying the mappings of such data structures onto structures of environments, including many different one-to-one and one-to-many mappings, and also provides fairly efficient implementation of the communication that can result through the use of these variables. The declaration of distributed data, including the definition and meaning of mapping functions, is discussed in Section 3.4.1. The use of distributed data, primarily remote accesses for communication, is discussed in Section 3.4.2.

3.4.1 Declaring Distributed Data -- Mapping Functions

distributed data declaration:

```
DECLARATOR ::=
    [ 'asynch' ] 'distributed' DECLARATOR ( '[' CONSTANT_EXPRESSION ']' ) + MAPPING .

MAPPING ::= 'map' ( 'all' | IDENTIFIER | ( ( IDENTIFIER IDENTIFIER ) || 'map' ) ) .
```

mapping function definition:

```
MAPPING_FUNCTION ::=
    'map' IDENTIFIER '=' ( '[' MAP_TYPE [ ALIGN ] ']' )+ .

MAP_TYPE ::= 'all' | 'compress' | BLOCK_MAP | WRAP_MAP .

BLOCK_MAP ::=
    'block' [ 'overlap' [ EXPRESSION] ] [ 'cross' 'axis' EXPRESSION ] .

WRAP_MAP ::= 'wrap' [ EXPRESSION ] .

ALIGN ::= 'align' 'axis' EXPRESSION .
```

Distributed data declarations follow standard C syntax, except that the keyword ''distributed'' precedes the name of the distributed variable, and the keyword ''map'' followed by the name of a mapping function follows it. An example is

float distributed A[N][N] map BlockRow;

The mapping function is either taken from DINO's library of predefined mapping functions, as is the case with ''BlockRow'' above, or is defined by a mapping function definition statement. The remainder of this subsection discusses mapping functions.

DINO provides predefined mapping functions for mapping one and two dimensional (hereafter ''1D'' or ''2D'') data structures onto 1D structures of environments, and 2D data structures onto 2D structures of environments. The predefined 1D to 1D mappings are

Block, Wrap, BlockOverlap.

They can be used to map any array of N variables onto any array of P environments, and work in a fairly obvious way. If N is a multiple of P, then ''Block'' maps the first N/P data elements onto the first environment and so on; if $N<P$ then the first N environments contain one element each and the last $N-P$ contain none, while if $N>P$ but N is not a multiple of P, then the first N mod P environments contain an extra element. ''Wrap'' maps each element i ($i=0,...,N-1$) to environment [i mod P]. ''BlockOverlap'' does a Block mapping and in addition, maps the first and last elements of each block (except elements 0 and N-1) to the next lower and higher environment, respectively. (One-sided overlaps and wider overlaps are also possible, see below).

The predefined mappings for mapping 2D (M x N) data structures to 1D (P) environment structures are

BlockRow, BlockCol, WrapRow, WrapCol, BlockRowOverlap, BlockColOverlap.

They work identically to the above 1D to 1D mappings except that rows or columns are used in the place of individual elements. For mapping 2D (M x N) data structures onto 2D (P x Q) environment structures, the predefined mapping functions are

BlockBlock, FivePoint, NinePoint.

"BlockBlock" is the cross-product of a block mapping of the first axis of the data structure onto the first axis of the environment structure with a block mapping of the second axis of the data structure onto the second axis of the environment structure, resulting in sub-arrays of size M/P x N/Q on each environment. "FivePoint" is the cross product of a BlockOverlap mapping in each direction; in the case when $M=P$ and $N=Q$ it reduces to the standard five point star (element $[i][j]$ mapped on to environments $[i-1][j]$, $[i+1][j]$, $[i][j]$, $[i][j-1]$, and $[i][j+1]$, and visa versa). "NinePoint" is a FivePoint mapping where in addition the corner elements are included in the overlap (in the $M=P$, $N=Q$ case these are $[i-1][j-1]$, $[i-1][j+1]$, $[i+1][j-1]$, $[i+1][j+1]$). Finally, the mapping keyword "all" maps all the elements of any dimensional data structure onto each environment of any dimensional structure of environments.

All these predefined mapping functions are special cases of DINO's general facility for defining mapping functions. Because this facility is rather complex, we will only give a general idea of it here and then give a number of examples; for more detail, see [RoW90]. The first part of the mapping function specifies how the axes of the data structure are matched to the axes of the environment structure, using the keywords "compress" (when the data structure has higher dimension, see below) and "align axis" (to override the default mapping of axis i to axis i). The second part specifies how a specific data axis is mapped onto a specific environment axis, using the keywords "block", "wrap", and "all". Finally, one may specify the width of wraps if greater than 1 ("EXPRESSION" in the WRAP_MAP rule), or the size and directions of overlaps in a block mapping ("overlap" and "cross axis").

For example, BlockRow is defined by

map BlockRow = [block][compress]

which says to block map the first axis of the data structure onto the first axis of the environment structure, and not to break up the second data axis. For WrapRow one replaces ''block'' by ''wrap'', while for BlockRowOverlap one replaces ''block'' by ''block overlap 1,1''. The ''1,1'' specifies one row of overlap to the left and right respectively, and can be replaced by any pair of nonnegative numbers. The column mappings are achieved by interchanging the order of the two expressions, e.g. ''[compress][block]''. BlockBlock is defined by ''[block][block]''; if one wants to map axes 0 and 1 of the data structure to axes 1 and 0 of the environment structure one uses ''[block align axis 1][block align axis 0]'' instead. FivePoint is ''[block overlap 1,1][block overlap 1,1]'' while NinePoint is ''[block overlap 1,1 cross axis 1][block overlap 1,1]''. In any of these cases an axis could instead be mapped ''wrap x'', resulting in a wrap mapping of that axis by blocks of width x (the default is x=1).

Implicitly associated with each mapping function is the specification of one ''home'' environment, and zero or more ''copy'' environments, for each element of the distributed data structure. These constructs have an effect when distributed data structures are parameters to composite procedures (see Section 3.3) and in remote accesses to distributed data (Section 3.4.2). If an element of a distributed data structure is mapped to only one environment, this is its home. If it is mapped to multiple environments, this must be done using either ''overlap'' or ''all''. With overlap, the home environment is the environment that the data element would be mapped to if the overlap term was omitted, while the additional environments that the overlap term causes it to be mapped to are its copy environments. (This concept is consistent with many uses of such mappings, for example in differential equation solvers based upon grids of points, where only one process produces new values of the shared grid point and the bordering processes consume these values.) If an element is mapped using ''all'', then we arbitrarily consider the lowest subscripted environment to be its home and the remainder to be copy environments, but this distinction is generally unimportant since such variables are usually either input-only parameters, or uniformly shared variables internal to the composite procedure. The semantics of home and copy environments also assure that DINO programs are deterministic in the default case (see Section 3.4.2).

Mapping function declarations can be placed either outside any environments, or inside the appropriate environment (but outside its functions and composite procedures). Distributed data may be declared at any level of a DINO program. Both follow standard scoping rules. The MAPPING portion of a distributed data declaration that is outside any environment must use the ''IDENTIFIER IDENTIFIER || map'' syntax, where the second identifier in each pair is an environment name, e.g. ''float distributed A[N][N] map grid BlockBlock map node BlockRow'', where grid and node are environment structures defined in Section 3.2 and 2.1.

3.4.2 Using Distributed Data

```
EXPRESSION ::=
    PRIMARY '#' [ '{' ENV_EXP [ 'from' EXPRESSION ] '}' ].

ENV_EXP ::= 'caller' | EXPRESSION .
```

There are three different ways to access a distributed variable. The first is as a parameter in a composite procedure call, as discussed in Section 3.3. The other two are by local and remote accesses, either reads or writes, in any standard C statement.

A local access of a distributed variable uses standard syntax and is the same as accessing any regular variable. The value of the variable is retrieved from, or stored into, the local copy. This implies that the variable being accessed must be mapped to the environment in which the access is made.

A remote access, either a read or a write, involves communication between environments. It is indicated by the distributed variable name, followed by a ''#'', optionally followed by an explicit specification of the set of environments to communicate with for this access. If this set is not provided explicitly, it is defined implicitly by the mapping function to be the set of all environments to which the distributed variable is mapped. The implicit environment specification is usually sufficient, and is considered preferable because it is more structured and because there is less likelihood of coding error.

We first describe the semantics of remote accesses in the case where the environment set is specified implicitly (by the associated mapping function). These accesses basically follow the convention that the home environment produces new values and the copy environments consume them.

If the remote access is a read and the current environment is a copy environment, then DINO looks for the first (least recent) value of that variable that has been received from the home environment but not yet utilized. (I.e., a message buffer is maintained, and the oldest message from the home environment with a value of that variable is used, and then removed from the buffer.) If no new value of that variable has been received from the home environment, the reading process blocks until one is received. When the new value is available, it is used to update the local copy of the variable, and then the remainder of the expression in which the remote read is located is evaluated, using the local copy. If the remote access is a read, the current environment is a home environment, and there are no associated copy environments, then the remote read is the same as a local read. (There is, however, some performance degradation.) If the current environment is a home environment and there are copy environments, or the current environment is neither a home nor a copy environment, then an implicit remote read is an error.

A remote write to a distributed variable using an implicit environment set works as follows. If the current environment is the home environment of that variable and there are associated copy environments, then the value being assigned to that variable is used to update the local copy (since the write occurs in an assignment context), and also is sent to all the copy environments. (If multiple environments are involved, this is done efficiently using a tree.) If the current environment is the home environment and there are no associated copy environments, then the remote write is treated as a local write (again, there is some performance degradation). If the current environment is not the home environment, then an implicit remote write is an error.

As example of remote reads and a remote write, consider a 2-dimensional smoothing algorithm where an N x M distributed array of data A is mapped onto an N x M array of virtual processors *grid*, with each $A[i][j]$ mapped onto $grid[i][j]$ (its home environment) and the 4 adjoining environments. (This is the mapping function FivePoint.) Then the statement

```
A[i][j]# = (A[i-1][j]# + A[i][j-1]# + A[i][j] + A[i][j+1]# + A[i+1][j]#)/5;
```

will receive values of the 4 adjoining elements of *A* from their home environments, calculate the average of these four values plus the local value, assign this new value to the local copy of *A* [*i*][*j*], and send it to the 4 adjoining environments.

If the programmer wishes to send or receive distributed variables in a different manner than by these implicit rules, a set of environments — ENV_EXP in the EBNF specification above — can be specified after the # sign. ENV_EXP uses the same syntax as was discussed for it in Section 3.3. For example, if the two dimensional array of environments *grid* is defined as above, then

```
grid[][] - grid[xid][yid] - grid[1][2]
```

specifies all the environments in the grid environment structure except the environment that the statement is executed in and *grid*[1][2]. In contrast to remote accesses with implicit environment sets, the distributed variable does not need to be mapped to the environment where the statement is executed.

A remote read with an explicit environment set is processed as follows. The process looks for the oldest unutilized value of that distributed variable that has been received from any node in the environment set, including itself if it is in the set, and blocks if no new value is available. Once a value is received, it is used in evaluating the remainder of the expression. (Note that the execution may be nondeterministic if there is more than one environment in the set.)

For a remote write with an explicit environment set, the new value of the variable is sent to all of the environments in the environment set, including itself if it is in the set. Note that explicit accesses to variables that are mapped to the local environment have different semantics than in the implicit case: they do generate messages, and do not automatically update the local copy or default to local accesses. In examples that we have considered, these semantics seemed appropriate.

After a remote read where the environment set was specified explicitly and contained more than one element, it may be unknown which environment the new value came from. In order to obtain this information, the "from" construct may be used. The keyword "from" is appended to the environment set

specification, and is followed by a variable of type ''envvar''. For example, the statements

```
envvar who;
   ...
w = x # {grid[][] from who};
   ...
y # {who} = z;
```

receive the value of x from some environment in the structure *grid*, assign the name of this environment to *who*, and later send the value of z to the environment that the value of x was received from.

All of the above discussion pertains to ''synchronous'' distributed variables, which are the default in DINO. Distributed variables can also be declared to be ''asynchronous'' by placing ''asynch'' before ''distributed'' in their declarations. The difference between synchronous and asynchronous variables is that a remote read to an asynchronous variable uses the most recent value and does not block. If several values of the variable have been received since the last remote read, then the most recently received value is used to update the local copy and the remaining values are discarded. If no new value has arrived since the last remote read, then the local copy is used. In keeping with these rules, an explicit remote read of an asynchronous variable requires that the variable be mapped to the current environment. The remaining semantics of remote accesses to asynchronous variables, including all the semantics for a remote write, are the same as for synchronous distributed variables.

By making the distinction between synchronous and asynchronous communication part of the distributed variable declaration, as opposed to a property of the statement that invokes communication, it is possible to transform a DINO program from a synchronous to an asynchronous variant by simply changing one or a few declarations. Clearly, the execution of such algorithms may be non-deterministic.

3.5. Reduction Functions.

```
PRIMARY ::=
    REDUCTION '(' EXPRESSION [ ',' EXPRESSION ] ')' '#' [ '{' ENV_EXP '}' ].

REDUCTION ::=
    'gsum' | 'gprod' | 'gmin' | 'gmindex' | 'gmax' | 'gmaxdex'.
```

One type of calculation that involves communication is so fundamental to parallel computation, and departs sufficiently from simple patterns of communication, that we have included it as a parallel language construct. This is a reduction operation, which involves performing a commutative operation (e.g. +, *, min, max) over one value from each environment, and often, returning the result to all the environments. DINO provides the reduction operators ''gsum'', ''gprod'', ''gmax'', and ''gmin'' that return the sum, product, maximum, or minimum of the arguments specified on each environment. For example, if *error* and *maxerror* exist on each environment in the current active set, then

maxerror = gmax(error)

assigns the maximum of all the local values of *error* to each local copy of *maxerror*. DINO also provides two reduction functions, ''gmaxdex and ''gmindex'', that take two parameters; the first is the value to be reduced and the second is (a pointer to) an integer index. When these functions return, the second parameter returns (points to) the index of the maximum (or minimum) value.

The reduction functions can be called with an explicit ''active set'' (ENV_EXP above) in the same way that a composite procedure or a remote data access can. The active set specifies which environments will participate in the reduction. The first argument of a one argument reduction may evaluate to a (sub)array, rather than a scalar, in which case the reduction operation is performed individually over each component of the (sub)array.

It is implicit in a reduction call that barrier synchronization is involved, and that all participating environments must reach the call, otherwise execution is blocked. The environments do not necessarily have to execute identical lines of code, but simply the same operation with matching parameter types. All reductions are implemented using efficient, tree-based computation and communication.

3.6. Subarrays and Ranges.

PRIMARY ::= PRIMARY ('[]' | '[<' EXPRESSION ',' EXPRESSION '>]'.

To use most distributed memory multiprocessors efficiently, it is important to send messages consisting of blocks of data, rather than multiple messages consisting of single data elements, whenever possible. In order to efficiently move blocks of data between environments, DINO provides the ability to specify subarrays, and to use arrays and subarrays in simple assignment statements, which may include remote reads and writes. (C provides no subarray facilities.)

A rectangular subsection of an array is specified by giving the first and the last element for each axis, e.g. "*A* [<*i* ,*j* >][]". The index numbers are separated by a comma and enclosed in angle brackets. The default value, specified by "[]", is all elements of the array along the indicated axis.

Arrays or subarrays, including remote accesses to them, may be used in assignment statements of the form "*A* [<*I* 1,*J* 1>]...[<*In* ,*Jn* >] = *B* [<*K* 1,*L* 1>]...[<*Kn* ,*Ln* >]". The size and shape of the operands in the statement must be consistent, and the indices of the left and right operand are mapped to each other in the obvious manner. If the left operand is a remote access to a distributed array, then the target environments of each element depend on the mapping function. In this way an entire array can be distributed over a set of environments in a single assignment, with different portions possibly going to different environments. Conversely, if the right operand is a remote access to a distributed (sub)array, this (sub)array is gathered from one or more environments in a single assignment.

DINO also allows the use of arrays and subarrays as parameters for composite procedures and reduction functions, and allows ranges to be used in specifying environment sets. At present we have not made additions to C to allow arithmetic operations on (sub)arrays, or the use of (sub)arrays as parameters or return values to standard C functions, although we plan to do so shortly. Due to these limitations of C, remote reads or writes of arrays or subarrays cannot currently be natural, implicit parts of standard C statements (e.g. arithmetic operations or calls to C functions), as we would desire. Instead, they often end up being specified by fairly explicit communication statements like "*A* [*i*][]#=*A* [*i*][]","*A* [*i*][]#=*B* []", or their remote read counterparts, after new values of the distributed variable have been generated, or before they are to be used (see Example 4.2). In acknowledgement of these current limitations, we have added the macros

```
define SEND(v) v#=v
define RECV(v) v=v#
```

to DINO (see Example 4.2). While these clearly depart from the implicit communication philosophy of the language, this departure is, for the most part, a temporary one caused by the array limitations of C, and for now the macros add some clarity to DINO programs that make remote accesses to (sub)arrays. Note that this last discussion does not pertain to composite procedure parameters, which are perhaps the main source of communication in practice; here arrays or subarrays are permitted, and communication is natural and implicit.

4. Examples of DINO Programs and Critique of DINO Programming Constructs

Examples 4.1 - 4.3 contain complete DINO programs for three typical numerical computation kernels. These examples have been selected to illustrate the basic DINO programming constructs, and also to highlight some current limitations of the language. In this section we briefly discuss these examples, and some of the strengths and weaknesses of the language that they illustrate. All the examples have a simple structure, in particular one structure of environments and one composite procedure; a more complex example is given in [RSW89].

The first example, a matrix-vector multiplication with the matrix partitioned by rows, is coded with a structure of environments that contains the same number of environments, N, as there are rows in the matrix. As mentioned in Section 3.2, in the current implementation, if there are fewer than N processors, this would lead to multiprogramming the nodes and hence suboptimal efficiency. At present, a more efficient DINO program is obtained by changing the number of environments to the actual number of processors, P, changing *id* to i in the 3 executable lines of *MatVec*, and placing these lines inside the loop "(for i=*id**N/P ; i<(*id*+1)*N/P ; i++) { ... }". While this is not too objectionable, we include the N-fold parallel example to make the point that we consider this the most natural way to program such a data parallel algorithm. Later in this section we mention anticipated modifications to DINO that would make it possible to program in this fully data parallel style without sacrificing efficiency.

The second and third examples, a row-wise red-black algorithm for solving Poisson's equation on an N x N grid, and the LU factorization of an N x N matrix, are shown in the form that they would be coded for the current version of DINO, i.e. for a structure of P (presumably $< N$) environments. In the red-black algorithm, at each iteration, each processor calculates a new value of each of its odd numbered rows, based upon the current value of this row and the two adjacent even numbered rows, and then each processor calculates a new value of each of its even numbered rows, based upon the current value of this row and the two adjacent odd numbered rows. Thus the parallel algorithm maps the rows to the environments in blocks, and maps a copy of each border row onto the adjacent environment. (We have assumed that N is a multiple of P, since the programmer generally can control this in such algorithms.) The LU factorization algorithm performs N iterations, each of which calculates a pivot and multipliers based upon column k (the iteration number), and then uses this information to perform eliminations in all columns whose index is greater than k. Thus it is implemented efficiently on a distributed memory multiprocessor by partitioning the matrix by columns and using a wrap mapping. (It is immaterial whether N is a multiple of P.)

Now we comment on the suitability of the DINO parallel programming constructs as illustrated by these examples. At an overall level, all three programs appear to us to be fairly natural and understandable. They certainly appear preferable to the same algorithms programmed using vendor-supplied constructs. For example, they are each about 2-3 times shorter than the code for the same algorithms using Intel iPSC hypercube constructs, and far easier to write and understand, while still executing nearly as efficiently as the programs using iPSC constructs (see Section 5).

Examined in more detail, these examples illustrate the main DINO features, namely structures of environments, composite procedures, distributed data declarations, and the use of distributed data for communication. To us, the first three seem quite satisfactory for these examples. The only significant shortcoming that we see was discussed above, namely that for efficiency reasons the number of environments currently should not exceed the number of processors. This limitation hardly complicates the coding of these three examples but it would be more natural to eliminate it. We return to this issue below.

The use of distributed data for communication occurs in two ways, using distributed data as parameters to composite procedures, and through remote (#) reads and writes. The former is natural and entirely implicit, and in programs like Example 4.1 where it is the only form of communication, we feel communication is handled nicely.

We consider the second form of communication, remote accesses, to be the least satisfactory current aspect of DINO. First of all, it might be argued that the programmer should have to make no distinction between local and remote accesses, that instead these should be distinguished by the compiler or possibly the run-time system. This is done, for example, in C* [RSt87], which is targeted for SIMD machines (where one also may assume that communication is fast relative to computation), and in Kali [MvR89], which uses a restricted form of an SPMD programming model (see Section 1). It seems to us that a programming style for parallel, distributed computation that allows no distinction between local and remote accesses, and still results in efficient code, is only possible if the model of parallel computation is restricted considerably, as in these languages. In contrast, to never explicitly distinguish between local and remote accesses in a language like DINO, which supports entirely general SPMD (and functionally parallel) computation and strives for efficient communication in an environment where communication is presumed to be relatively slow, would be extremely difficult if not impossible. It would require extremely general data dependency analysis to determine when data should be sent and received, as well as optimizations for communication efficiency such as bundling subarrays into single messages and prefetching (sending messages as soon as new values are available rather than requesting them when they are required). Thus we believe it is necessary to include some facility for explicit remote accesses in a general purpose language for distributed computation like DINO. What we are investigating is also including a more restricted, SIMD-like composite procedure as an option in DINO, and removing the requirement for the programmer to distinguish between local and remote accesses in such procedures. Instead, the partial SIMD semantics would allow the compiler to perform the analyses mentioned above. Our preliminary ideas on this topic are discussed in [RSW90]. This new option would be applicable to many numerical algorithms, such as Examples 4.2 and 4.3, and would remove # accesses from them entirely. It is also the main ingredient that would allow the programmer to express these algorithms at their natural

N-fold level of parallelism and then enable the compiler to contract them to efficient P-fold algorithms.

The second shortcoming of remote accesses in the current version of DINO is the cumbersome nature of some of them. While the changes mentioned above would eliminate remote accesses from many DINO programs, in some programs they would remain and we would like them to be as simple and natural as possible. Examples 4.2 and 4.3 have been chosen to illustrate two main problems with expressing remote accesses. First, the "SEND(subarray)" and "RECV(subarray)" type accesses that we discussed at the end of Section 3.5 are used in Example 4.2. These are only required because C does not have facilities for using subarrays in arithmetic operations or as parameters. Our compiler already has most of the hooks for such facilities and we plan to incorporate them in the future. At this point, one could encapsulate the communication entirely in statements like

```
solveRow (U[i-1][]#,U[i][]#,U[i+1][])
```

which seem reasonably satisfactory to us. Secondly, in the broadcast in Example 4.3, it would be nice not to require the explicit environment set { *node*[] }, but this is currently necessary due to the semantics of distributed variables which permit implicit remote writes only from a single, home environment. We are considering modifications to the language that would alleviate this type of problem while preserving determinism. One possibility is to allow the home environment to be dynamically redefined within the program.

Finally, the Examples 4.2 and 4.3 point out a relatively minor addition to DINO that would be quite convenient. When distributed data structures are used, it would be nice to have library functions that return the indices of that data structure that are mapped onto the current environment. For instance in Example 4.2, "first(U, 0)" and "last(U, 0)" would return the first and last indices of the first (number 0) axis of U that are mapped onto the current environment; this is the information that is currently specified by the program in the variables *firstrow* and *lastrow*.

Example 4.1

Matrix-Vector Multiplication

```
#define N 512
#include <stdio.h>
#include <dino.h>

environment node [N:id]
  {
  /* Row-wise Parallel Algorithm to Calculate  a <-- M*v  : */

  composite MatVec (in M, in v, out a)
    float distributed M[N][N] map BlockRow;
    float distributed v[N] map all;
    float distributed a[N] map Block;
    {
    int j;

    /* a[id] <-- dot product of (row id of M) and v */
    a[id] = 0;
    for (j=0; j<N; j++)
      a[id] += M[id][j] * v[j];
    }
  }

environment host
  {
  main ()
    {
    long int i,j;
    float Min[N][N];    /* Matrix Multiplicand */
    float vin[N];       /* Vector Multiplicand */
    float aout[N];      /* Vector Answer */

    /* Input the Data */
    for (i=0; i<N; i++) {
      vin[i] = i;
      for (j=0; j<N; j++)
        Min[i][j] = i*j;
      }

    /* Call the Composite Procedure */
    MatVec (Min[][], vin[], aout[])# ;

    /* Print the Results */
    for (i=0; i<N; i++)
      (void) printf("%.2f0, aout[i]);
    }
  }
```

Example 4.2

Row-Wise Red Black Algorithm for Poisson's Equations

```
#define N 512
#define P 32
#include <stdio.h>
#include <math.h>
#include <dino.h>

environment node [P:id]
  {
  double d[N], s[N];
  int needfactor = 0;

  void solveRow (U,i)     /*  solves  tridiag (-1,4,-1) * U[row_i] = (U[row_i-1]+U[row_i+1])  */

    double distributed U[N][N] map BlockRowOverlap;
    int i;
    {
    int j;
    double r = 1.0;       /* relaxation constant, can be different constant */
    double new[N];

    if (needfactor == 0)
       {
       /* factor tridiag (-1,4,-1) */
       needfactor = 1;
       d[1] = 2;
          for (j=2; j<N-1; j++)
            {
            s[j] = - 1/d[j-1];
            d[j] = sqrt (4 - s[j]*s[j]);
            }
       }

    /* forward solve  (solve for elements 1...N-1 of row i, add U[i][0], U[i][N-1] to right hand side)  */
    new [1] = (U[i-1][1] + U[i+1][1] + U[i][0])/d[1];
    for (j=2; j<N-2; j++)
       new [j] = (U[i-1][j] + U[i+1][j] - s[j]*new[j-1])/d[j];
    new [N-2] = (U[i-1][N-2] +U[i+1][N-2] +U[i][N-1] -s[N-2]*new[N-3])/d[N-2];

    /* backward solve */
    new [N-2] = (new[N-2])/d[N-2];
    for (j=N-3; j > 0; j=j-1)
       new [j] = (new[j] - s[j+1]*new[j+1])/d[j];

    /* relaxation */
    for (j=1; j<N-1; j++)
       U[i][j] = r*new[j] + (1-r)*U[i][j];
    }
```

```
/* Parallel Algorithm For Row-Wise Red Black, N/P Rows per Processor : */
composite solver (U)

   double distributed U[N][N] map BlockRowOverlap;
    /* partitions U by blocks of rows among processors, with each border row mapped to 2 processors */

   {
   int firstrow = id ? (id * N/P) : 1;                    /* indices of first and last rows */
   int lastrow = (id == (P-1)) ? N-2 : ((id+1) * (N/P) - 1);   /* in each block */
   int firsteven = firstrow%2 ? firstrow + 1: firstrow ;
   int firstodd = firstrow%2 ? firstrow : firstrow + 1;
   int k, i;

   for (k=0; k<100; k++)
      {
      for (i=firstodd; i <= lastrow; i=i+2)       /* update odd rows in each block */
         {
         if ((i==firstrow) && (i != 1) && (k != 0))
            RECV(U[i-1][]);     /* DINO Macro, see end of Section 3.6 */
         if ((i==lastrow) && (i != N-2) && (k != 0))
            RECV(U[i+1][]);
         solveRow (U,i);
         if (((i==firstrow) && (i != 1)) || ((i==lastrow) && (i != N-2)))
            SEND(U[i][]);      /* DINO Macro, see end of Section 3.6 */
         }

      for (i=firsteven; i <= lastrow; i=i+2)       /* update even rows in each block */
         {
         if ((i==firstrow) && (i != 1) && (k != 0))
            RECV(U[i-1][]);     /* DINO Macro, see end of Section 3.6 */
         if ((i==lastrow) && (i != N-2) && (k != 0))
            RECV(U[i+1][]);
         solveRow (U,i);
         if (((i==firstrow) && (i != 1)) || ((i==lastrow) && (i != N-2)))
            SEND(U[i][]);      /* DINO Macro, see end of Section 3.6 */
         }
      }
   }
}


environment host
   {
   main ()
      {
      float Uh [N][N]; /* input and output to ''solver'' */
      long int i,j;

      /* Input the Data (Uh) */
      for (i=0; i<N; i++)
         for (j=0; j<N; j++)
            Uh[i][j] = i*j;
      for (i=1; i<N-1; i++)
         for (j=1; j<N-1; j++)
```

```
        Uh[i][j] = Uh[i][j] - .10;

    solver (Uh[][])# ;  /*  Composite  Procedure Call  */

    /*  Printing the Results (Uh)  */
    for (i=0; i<N; i++)
        {
        for (j=0; j<N; j++)
            (void) printf("%.2f ", Uh[i][j]);
        (void) printf("0);
        }
    }
}
```

Example 4.3

LU Factorization

```
#define N 512
#define P 32
#include <stdio.h>
#include <math.h>
#include <dino.h>


environment node [P:id]
  {
  /*  Parallel LU Factorization Algorithm :  */
  composite lufactor (A)

    float distributed A[N+1][N] map WrapCol;
     /*  does wrap (cyclic) mapping of columns of A to the environments  */
    {
    float distributed mult [N+1] map all;  /* places copy of this vector in each environment  */
    int i, j, k, pivrow;
    float piv, temp;

    for (k=0; k<N-1; k++)
      {
      if ((k%P) == id)  /*  my environment contains pivot column for this iteration  */
        {
        /* select pivot and swap in pivot column */
        piv = A[k][k];  pivrow = k;
        for (i=k+1; i<N; i++)
          if ((fabs(A[i][k])) > piv)  {
            piv = A[i][k];  pivrow = i;  }
          if (pivrow != k)  {
            temp = A[k][k]; A[k][k] = A[pivrow][k]; A[pivrow][k] = temp;   }

        /* calculate multipliers and broadcast them and pivot row number */
        A[N][k] = pivrow; /* N+1st row of A will contain pivots */
        for (i=k+1; i<N; i++)
          A[i][k] = A[i][k]/A[k][k];
        mult [<k+1, N>] # { node[] } =  A [<k+1, N>][k];  /*  broadcast  */

        }

      /* receive pivot information */
      mult [<k+1, N>] = mult [<k+1, N>] # { node[(k%P)] };
```

```
        /* eliminate in your columns greater than k */
        pivrow = mult[N];
        for (j = id; j < N; j=j+P)
          if (j>k)
            {
            if (pivrow != k) {
              temp = A[k][j]; A[k][j] = A[pivrow][j]; A[pivrow][j] = temp; }
            for (i=k+1; i<N; i++)
              A[i][j] -= mult[i]*A[k][j];
            }
        }
      }
    }


environment host
  {
  main ()
    {
    float A [N+1][N];  /* input and output to ''lufactor''", last row will contain pivots */
    long int i,j;

    /* Input A */
    for (i=0; i<N; i++)
      for (j=0; j<N; j++)
        A[i][j] = (i+1)*(j+1);
      for (i=0; i<N; i++)
        A[i][i] += 1;

    lufactor (A[][])# ;      /* Composite Procedure Call */

    /* Print the Results */
    for (i=0; i<N; i++)
      {
      for (j=0; j<N; j++)
        (void) printf("%.2f ", A[i][j]);
      (void) printf("0);
      }
    }
  }
```

5. Current Status and Future Research Directions

We have written a full compiler for DINO. It was implemented using the compiler generation tools of [GHKSW89]. The compiler produces C code augmented by the low level parallel instructions of the target parallel machine. The initial version runs on the Intel iPSC1 and iPSC2 hypercubes, and on the simulators for these machines. The production of machine-dependent statements in the compiler has been isolated, however, so that porting it to other distributed memory multiprocessors is not expected to be difficult. The compiler is available at no charge to universities and research laboratories.

We have compared the lengths and execution times of DINO programs, including the examples in Section 4, to identical parallel algorithms coded as similarly as possible in C using the Intel iPSC2 primitives. In general, the DINO programs are 2-3 times shorter than the programs using iPSC primitives. Their runs times on the iPSC2 range from within 1% of the iPSC2 primitive versions to roughly double the iPSC version. Whenever the degradations are greater than 10-20%, our measurements have shown that they are due to those communication features that, for expediency, were implemented via a run-time library rather than in the compiler. This was done for all parameter distribution via distributed data, and for many communications generated via remote (#) accesses to distributed data. To demonstrate that communication via remote accesses could be made efficient, we implemented the most common remote access mechanisms, namely those using the *BlockRow* and *BlockCol* mappings, and all remote accesses that use explicit environment sets (e.g. Example 4.3), in the compiler. In almost all cases, these require less than 5% more time than the same communications generated using iPSC primitives. The remaining remote accesses are currently generated via the run-time library, and generally require about twice as much time as the same communications generated using iPSC primitives. High efficiency in parameter distribution was considered less important for a first version of the compiler, since parameter distribution constitutes a relatively small part of most programs, and thus it is handled in the run-time library. Parameter distributions using distributed data generally are at most twice as expensive as the same communications using iPSC primitives, however certain cases are up to 5-7 times slower. (Among the default mappings, the one slow case is the wrap column mapping, where each data access is noncontiguous; the offset

calculation is much slower when handled by a general facility than by one tailored to the mapping.)

In summary, our measurements show that a communication intensive DINO program that uses the remote accesses that have been implemented in the compiler is at most 10% slower than the equivalent program using low-level primitives, unless the computation is dominated by parameter distribution. We estimate that at most 2-3 additional months of compiler development work would be required to implement all the remaining communication features, which are currently handled by the run-time library, in the compiler, and that all DINO programs would then require at most 10-30% more time than equivalent programs coded using low-level primitives. We feel this indicates that our high-level programming approach can be used without significantly hindering efficiency.

The development of the DINO language and compiler has given us a platform from which we wish to explore a number of interesting research issues. Some, involving language features and compiler optimizations, were discussed briefly in Section 4. These include the ability to contract processes automatically and efficiently, that is, write a DINO program for N (or some other function of the problem size) virtual processors, and have the compiler transform this program into an efficient program whose number of processes is equal to the number of available processors. This research problem is actually just a subproblem of a main research thrust, which is the exploration of issues that arise in expressing parallel algorithms for large, complex applications. These algorithms may involve multiple phases, and different data mappings and virtual parallel machines may be appropriate at different phases. A key ingredient in providing efficient support for such applications will be the ability to embed parallel algorithms within other parallel algorithms; then, depending on the level of embedding, a parallel algorithm may be contracted to anywhere from P to 1 actual processes. Thus the contraction problem is a key portion of the large scale application problem. A key ingredient in our approach to solving this problem is the provision of partially-SIMD composite procedures, which greatly facilitate automatic contraction, and also allow most remote accesses to become transparent to the user, as was discussed briefly in Section 4. Our preliminary ideas on these issues are presented in [RSW90].

In conjunction with this research into large scale parallel programming, we are collaborating with several scientists in using DINO and assessing its strengths and weaknesses. We are also beginning to use DINO in parallel programming instruction at the graduate and undergraduate level. Other related, longer range research issues we are considering include the provision of dynamic distributed data structures and virtual parallel machines in a high level distributed programming language, and visual interfaces to high level distributed programming systems.

References

[CK88] Callahan, D. and Kennedy, K., ''Compiling Programs for Distributed-Memory Multiprocessors'', *Journal of Supercomputing* 2, 1988, pp. 151-169.

[GCC85] Gelernter, D., Carriero, S., Chandran, S., and Chang, S., ''Parallel Programming in Linda'', *Proceedings of the 1985 Conference on Parallel Processing,* IEEE Press, 1985, pp. 255 - 63.

[GHKSW89] Gray, R.W., Heuring, V.P., Krane, S.P., Sloane, A.M., and Waite, W., ''Eli: A Complete, Flexible Compiler Construction System'', University of Colorado at Boulder Electrical and Computer Engineering Technical Report SEG 89-1-1, June 1989.

[HWW89] Hamey, L.G., Webb, J.A., and Wu, I.-C., ''An Architecture Independent Programming Language for Low-Level Vision'', *Journal of Computer Vision, Graphics, and Image Processing* 48, 1989, pp. 246-264.

[In87] Intel iPSC Programmer's Reference Guide, March, 1987, Number 310612-002.

[Jo87] Jordan, H., ''The Force'', *The Characteristics of Parallel Algorithms,* Jamieson, L., Gannon, D., and Douglass, R., Eds., MIT Press, 1987, pp. 395 - 436.

[KR78] Kernighan, B. and Ritchie, D., *The C Programming Language,* Prentice-Hall, Englewood Cliffs, N.J., 1978.

[KMvR89] Koelbel, C., Mehrotra, P., and Van Rosendale, J., ''Supporting Shared Data Structures on Distributed Memory Architectures'', to appear in *Proceedings of Conference on Principles and Practice of Parallel Processing,* March 1990.

[KS85] Kuehn, J.T., and Siegel,H.J., ''Extensions to the C Programming Language for SIMD/MIMD Parallelism'', *Proceedings of the 1985 International Conference on Parallel Processing,* IEEE Press, 1985, pp. 232 - 235.

[LWL85] Li, H., Wang, C., and Lavin, M., ''Structured Process'', *Proceedings of the 1985 International Conference on Parallel Processing,* IEEE Press, 1985, pp. 247 - 254.

[MvR89] Mehrotra, P., and Van Rosendale, J., ''Compiling High Level Constructs to Distributed Memory Architectures'', *Proceedings of Fourth Conference on Hypercubes, Concurrent Computers, and Applications,* Vol. 2, 1989, pp. 845 - 852.

[Pr87] Pratt, T., ''The Pisces 2 Parallel Programming Environment'', *Proceedings of the 1987 International Conference on Parallel Processing,* IEEE Press, 1987, pp. 439 - 45.

[Re84] Reeves, A.P., ''Parallel Pascal: An Extended Pascal for Parallel Computers'', *Journal of Parallel and Distributed Computing* 1, 1984, pp. 64-80.

[RSt87] Rose, J. and Steele, G., ''C*: An Extended C Language For Data Parallel Programming'', Pl85-7, Thinking Machines Corp., 1987.

[**RSc87**] Rosing, M., and Schnabel, R.B.,, "An Overview of Dino -- A New Language for Numerical Computation on Distributed Memory Multiprocessors", *Proceedings of Third SIAM Conference on Parallel Processing for Scientific Computation*, 1987, pp. 312-316.

[**RSW88**] Rosing, M. Schnabel, R.B., and Weaver, R.P., "Dino : Summary and Examples", *Proceedings of Third Conference on Hypercube Concurrent Computers and Applications*, 1988, pp. 472-481.

[**RSW89**] Rosing, M. Schnabel, R.B., and Weaver, R.P., "Expressing Complex Parallel Algorithms in DINO", *Proceedings of Fourth Conference on Hypercubes, Concurrent Computers, and Applications*, Vol. 1, 1989, pp. 553 - 560.

[**RSW90**] Rosing, M. Schnabel, R.B., and Weaver, R.P., "Massive Parallelism and Process Contraction in DINO", University of Colorado at Boulder Computer Science Technical Report CU-CS-467-90, to appear in *Proceedings of Fourth SIAM Conference on Parallel Processing for Scientific Computation*, December 1989.

[**RoW90**] Rosing, M., and Weaver, R.P., "Mapping Data to Processors in Distributed Memory Computers", to appear in *Proceedings of Fifth Conference on Distributed Memory Concurrent Computers*, April 1990.

[**RuW88**] Ruppelt, Th., and Wirtz, G., "From Mathematical Specifications to Parallel Programs on a Message Based System", *Proceedings of the 1988 International Conference on Supercomputing*, ACM, 1988, pp. 108 - 118.

[**SBB87**] Scott, L., Boyle, J., and Bagher, B., "Distributed Data Structures for Scientific Computation", *Proceedings of the Second Conference on Hypercube Multiprocessors*, 1987, pp. 55 - 66.

[**SSS88**] Seitz, C.L., Seizovic, J., and Su, W.-K., "The C Programmer's Abbreviated Guide to Multicomputer Programming", Caltech Computer Science Technical Report Caltech-CS-TR-88-11, 1988.

[**So90**] Socha, D.G., "Spot: A data parallel language for iterative algorithms", University of Washington Department of Computer Science Technical Report TR 90-03-01, 1990.

[**WG84**] Waite, W. and Goos, G., *Compiler Construction*, Springer-Verlag, 1984.

Appendix A

EBNF SPECIFICATION FOR DINO EXTENSIONS TO C

Note: All non-terminals that are not defined in this appendix are from the C language syntax definition in [KR78].

This EBNF specification uses the following notation:

()	**precedence grouping**
[]	**optional**
*	**zero or more**
+	**one or more**
|	**alternatives**
::=	**is replaced by**
||	**one or more separated by**

Terminals are enclosed in single quotes
Non-Terminals are in capitals

Program:

PROGRAM ::= (ENVIRONMENT | MAPPING_FUNCTION | DATA_DEFINITION)+ .

Environments:

ENVIRONMENT ::=
'environment' IDENTIFIER DIMENSION* '{' EXTERNAL_DEFINITION+ '}' .

DIMENSION ::= '[' EXPRESSION [':' IDENTIFIER] ']' .

EXTERNAL_DEFINITION ::=
FUNCTION_DEFINITION | DATA_DEFINITION | MAPPING_FUNCTION .

Composite Procedure Declarations:

FUNCTION_DEFINITION ::=
'composite' IDENTIFIER '(' [COMP_PARAMETER_LIST] ')' FUNCTION_BODY .

COMP_PARAMETER_LIST ::= (['in' | 'out'] IDENTIFIER) || ',' .

Composite Procedure Call:

STATEMENT ::=
    IDENTIFIER '(' [EXPRESSION_LIST] ')' '#' ['{' ENV_EXP '}'] ['::' STATEMENT] .

ENV_EXP ::= EXPRESSION .

Distributed Data Declaration:

DECLARATOR ::=
    ['asynch'] 'distributed' DECLARATOR ('[' CONSTANT_EXPRESSION ']')+ MAPPING .

MAPPING ::= 'map' ('all' | IDENTIFIER | ((IDENTIFIER IDENTIFIER) || 'map')) .

Distributed Data Use:

EXPRESSION ::=
    PRIMARY '#' ['{' ENV_EXP ['from' EXPRESSION] '}'] .

ENV_EXP ::= 'caller' | EXPRESSION .

Subarrays and Ranges:

PRIMARY ::= PRIMARY ('[]' | '[<' EXPRESSION ',' EXPRESSION '>]' .

Mapping Functions:

MAPPING_FUNCTION ::=
    'map' IDENTIFIER '=' ('[' MAP_TYPE [ALIGN] EXPANSION* ']')+ .

MAP_TYPE ::= 'all' | 'compress' | BLOCK_MAP | WRAP_MAP .

BLOCK_MAP ::=
    'block' ['overlap' [EXPRESSION]] ['cross' 'axis' EXPRESSION] .

WRAP_MAP ::= 'wrap' [EXPRESSION] .

ALIGN ::= 'align' 'axis' EXPRESSION .

EXPANSION ::= 'expand' 'axis' EXPRESSION .

Reduction Functions:

PRIMARY ::=
    REDUCTION '(' EXPRESSION [',' EXPRESSION] ')' '#' ['{' ENV_EXP '}'].

REDUCTION ::=
    'gsum' | 'gprod' | 'gmin' | 'gmindex' | 'gmax' | 'gmaxdex'.

A Note on C Syntax:

In standard C, a PROGRAM consists of one or more EXTERNAL_DEFINITIONs, each of which can be a FUNCTION_DEFINITION or a DATA_DEFINITION. DINO complicates this somewhat by adding environments.

C DATA_DEFINITIONs are built from DECLARATION_SPECIFIERs (e.g., int, struct A { int B; char C}, etc.), followed by DECLARATORs, followed by INITIALIZERs. The DECLARATOR is an IDENTIFIER, optionally nested inside one or more "", "()", or "[]" to designate respectively a pointer to, a function returning, or an array of. DINO allows a single distributed declaration in this nesting to designate a particular DATA_DEFINITION as distributed.*

Ordinary C FUNCTION_DEFINITIONs are built from an optional TYPE_SPECIFIER, a FUNCTION_DECLARATOR (which includes the parameter list) and a FUNCTION_BODY (which includes the parameter declarations). DINO uses a simpler syntax for composite procedures.

Certain types of C EXPRESSIONs are called PRIMARYs to distinguish then from the larger class of all EXPRESSIONs.